View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 149 (2006) 71–90

www.elsevier.com/locate/entcs

# Incremental Verification for On-the-Fly Controller Synthesis

David J. Musliner and Michael J. S. Pelican[1,2]

*Honeywell Laboratories*
*Minneapolis, MN, USA*

Robert P. Goldman[3]

*SIFT, LLC*
*Minneapolis, MN, USA*

**Abstract**

The CIRCA system automatically synthesizes hard real-time discrete event controllers from plant and environment descriptions. CIRCA's automatically-synthesized controllers provide guaranteed real-time performance and safety preservation in adversarial, non-closed-world domains. By separating controller construction from formal controller verification, CIRCA makes controller synthesis decisions in a time-abstract state space that is quite compact. However, controller verification requires a more complete consideration of time, to make real-time performance guarantees. By retaining information between verifications of partial controllers during the controller synthesis process, the *incremental verification* methods that we present here dramatically reduce the complexity of controller synthesis. We provide formal characterizations of our incremental verification technique and performance results demonstrating up to a 97% reduction in controller synthesis time using these methods.

*Keywords:* Incremental Verification, Controller Synthesis, Model Checking, Planning.

## 1 Introduction

As interest grows in deploying autonomous systems for mission-critical applications such as controlling spacecraft and military robots, research-grade

[1] Email: david.musliner@honeywell.com
[2] Email: mike.pelican@honeywell.com
[3] Email: rpgoldman@sift.info

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2005.07.027

autonomous control systems are under increasing pressure to provide some level of performance guarantees or assurances of correctness. In these domains, correctness involves not just ensuring that a logical or appropriate action is taken, but that it is taken at an appropriate and logical time [14]. Researchers are investigating various approaches including using formal verification methods to check planner code [18], and tying planning and execution semantics to formal automaton models [9].

The Cooperative Intelligent Real-Time Control Architecture (CIRCA) emphasizes guaranteed timely and logical performance by using formal verification methods to check that the plans it will execute meet strict logical safety and real-time response requirements [12,5]. Rather than verifying the planner code itself, CIRCA incorporates on-line verification of the plans its planner builds. This approach allows us to use an arbitrarily complex planning engine, including unguaranteeable methods such as domain-independent search heuristics, to produce plans which are then checked by a non-heuristic formal verification tool.

In prior publications we have described how CIRCA repeatedly maps its partial plans into timed automata during plan generation, and repeatedly uses a model checker to ensure that the growing plans meet safety and timing constraints. The key contribution of this paper is our *incremental verification* technique, which takes advantage of the relative stability of the planner's forward search to dramatically reduce plan verification time. As the planner makes new action choices for unplanned states, our CIRCA-Specific Verifier (CSV) retains partial verification traces and only generates extensions to those traces. We present the formal foundations, the algorithm, and performance results on a wide variety of domains showing dramatic performance improvements. While our discussion centers on how CIRCA uses these techniques, they can be more broadly applied to model checking in iterative generate-and-test settings such as offline or manual system design and verification.

## 2 CIRCA Background

CIRCA's Controller Synthesis Module (CSM) contains two main components of interest for this paper: the State-Space Planner (SSP) that reasons about time-abstract states to plan actions, and the CIRCA-Specific Verifier (CSV) that reasons about partial and complete plans to ensure that they meet logical and timing safety requirements. In this section, we briefly sketch these functional modules and describe an example problem that we will carry throughout the paper, to clarify how incremental verification works.

```
ACTION turn-on-main-engine              ;; Turning on the main engine
   PRECONDITIONS: '((engine off))
   POSTCONDITIONS: '((engine on))
   DELAY: <= 1

EVENT IRU1-fails           ;; Sometimes the IRUs break without warning.
   PRECONDITIONS: '((IRU1 on))
   POSTCONDITIONS: '((IRU1 broken))

;; If the engine is burning while the active IRU breaks,
;; we have a limited amount of time to fix the problem before
;; the spacecraft will go too far out of control.
TEMPORAL fail-if-burn-with-broken-IRU1
   PRECONDITIONS: '((engine on)(active-IRU IRU1) (IRU1 broken))
   POSTCONDITIONS: '((failure T))
   DELAY: >= 5
```

Fig. 1. Example transition descriptions given to CIRCA's CSM.

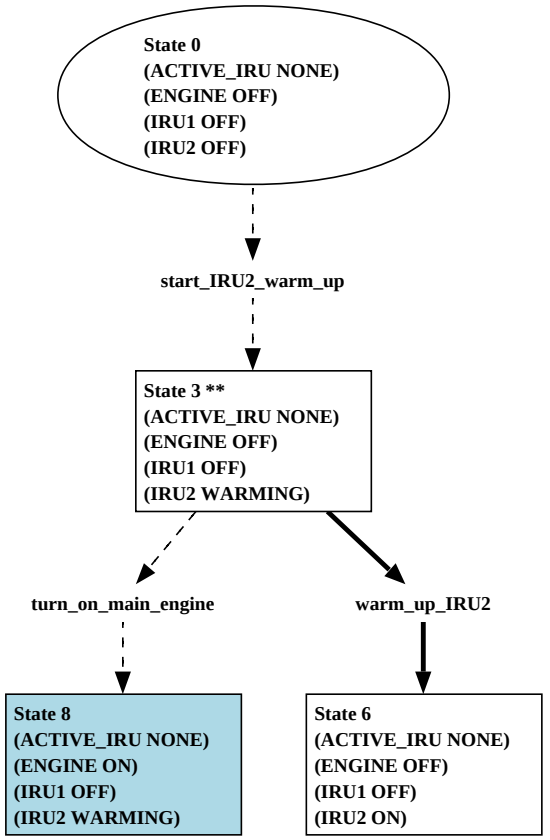


Fig. 2. The beginning of a state space plan for Saturn Orbit Insertion.

### *2.1 State Space Planning*

Unlike traditional AI planners, CIRCA reasons about uncontrollable processes including adversaries, and metric, continuous time. The SSP takes in a description of the processes in the system's environment, represented as a set of time-constrained transitions that modify the value of world features. Discrete states of the system are modeled as sets of feature-value assignments. Transitions have preconditions, describing when they are applicable, and bounded delays, capturing the temporal characteristics of controllable processes (i.e., actions) and uncontrollable processes (i.e., world dynamics). For example, Figure 1 shows three transitions from a CIRCA problem description for controlling the Cassini spacecraft during Saturn Orbital Insertion [4,13]. The transition descriptions, together with specifications of initial states, implicitly define the set of possible system states. The CSM is responsible for deciding, in each state, what action the system should take to maintain system safety and drive the system towards its goals. For example, Figure 2 illustrates a small portion of the Saturn problem's state space, after the CSM has made only its first few decisions about how to control the system. Note that this view of controller synthesis is somewhat different than that of Ramadge and Wonham [16], where the task is to choose which actions to disable in each state, rather than which preferred action to take in each state.

The CSM reasons about both controllable and uncontrollable transitions:

**Action transitions** represent actions selected by CIRCA; the SSP's objective is to assign an action to each reachable state. In Figure 2, a dashed arrow shows that the system has chosen the action `start_IRU2_warm_up` in the initial state zero. The special 'do nothing' action "no_op" can be assigned. Associated with each action is a worst case execution time, an *upper bound* on the delay before the action occurs.

**Temporal (uncontrollable) transitions** represent uncontrollable processes. Associated with each temporal transition is a *lower bound* on its delay. If the preconditions hold true for at least this time, the transition may fire and enforce its postconditions. If a temporal transition leads to an undesirable state, the CSM may plan an action to *preempt* the temporal by ensuring that the action will definitely occur before the temporal could possibly occur. Transitions whose lower bound is zero are referred to as *events*, and are handled specially for efficiency reasons. Transitions whose postconditions include the distinguished proposition `(failure T)` are called *temporal transitions to failure* (TTFs).

**Reliable temporal transitions** represent continuous processes that may need to be employed by the CIRCA agent. Reliable temporal transitions have both upper and lower bounds on their delays. For example, when CIRCA

turns on an Inertial Reference Unit it initiates the process of warming up that equipment; the process will definitely complete if it is continued without interruption for some time, as shown by the solid arrow leaving state 3 in Figure 2.

Note that each transition is an implicit description of many transitions in an automaton model. Each of these transitions is enabled in any discrete state that satisfies its preconditions, and disabled everywhere else.

**Algorithm 1 (Controller Synthesis).**

(i) *Choose a state from the set of reachable states (at the start of state space planning, only the initial states are reachable).*
(ii) *For each uncontrollable transition enabled in this state, choose whether or not to preempt it. Transitions that lead to failure states* must *be preempted.*
(iii) *Choose a single control action or* `no-op` *for this state.*
(iv) *Invoke the verifier to confirm that the (partial) controller is safe.*
(v) *If the controller is* not *safe, use information from the verifier to direct backjumping and goto step i.*
(vi) *If the controller* is *safe, recompute the set of reachable states.*
(vii) *If there are no "unplanned" reachable states (reachable states for which a control action has not yet been chosen), terminate successfully.*
(viii) *If some unplanned reachable states remain, loop to step i.*

As shown in Algorithm 1, the CSM search algorithm maintains the decisions that have been made, along with the potential alternatives, on a search stack. The algorithm makes decisions at two points: step ii and step iii.

The SSP uses the verifier to confirm both that failure is unreachable *and* that all the chosen preemptions will be enforced. The SSP uses the verifier module after each assignment of a control action (see step iv). This means that the verifier will be invoked before the controller is complete. At such points we use the verifier as a conservative heuristic by treating all unplanned states as if they are "safe havens." Unplanned states are treated as absorbing states of the system, and any verification traces that enter these states are regarded as successful. Note that this process converges to a sound and complete verification when the controller synthesis process is complete. When the verifier indicates that a controller is *unsafe*, the SSP will query it for a path to the distinguished failure state. The set of states along that path provides a set of candidate decisions to revise, as discussed in [6].

## 3 Formal Underpinnings

In this section, we provide a mathematical description of a controller graph and briefly introduce the corresponding timed automaton model and algorithms used for formal safety verification.

The search described by Algorithm 1 is conducted to create a controller graph:

**Definition 3.1** [CIRCA controller graph] $\mathcal{P} = \langle S, E, \vec{F}, \vec{V}, \phi, I, T, \iota, \eta, p, \pi \rangle$ where

(i) $S$ is a set of states.
(ii) $E$ is a set of edges.
(iii) $\vec{F} = [f_0 ... f_m]$ is a vector of features (in a purely propositional domain, these will be propositions).
(iv) $\vec{V} = [\mathcal{V}_0 ... \mathcal{V}_m]$ is a corresponding vector of sets of values ($\mathcal{V}_i = \{v_{i0} ... v_{ik_i}\}$) that each feature can take on.
(v) $\phi : S \mapsto \vec{V}$ is a function mapping from states to unique vectors of value assignments.
(vi) $I \subset S$ is a distinguished subset of initial states.
(vii) $T = U \cup A$ is the set of transitions, made up of an uncontrollable ($U$) subset, the temporals and reliable temporals, and a controllable ($A$) subset, the actions. Each transition, $t$, has an associated delay ($\Delta_t$) lower and upper bound: $\text{lb}(\Delta_t)$ and $\text{ub}(\Delta_t)$. For temporals $\text{ub}(\Delta_t) = \infty$, for events $\text{lb}(\Delta_t) = 0, \text{ub}(\Delta_t) = \infty$.
(viii) $\iota$ is an interpretation of the edges: $\iota : E \mapsto T$.
(ix) $\eta : S \mapsto 2^T$ is the *enabled* relationship — the set of transitions enabled in a particular state.
(x) $p : S \mapsto A \cup \epsilon$ (where $\epsilon$ is the "action" of doing nothing) is the actions that the SSP has planned. Note that $p$ will generally be a partial function.
(xi) $\pi : S \mapsto 2^U$ is a set of preemptions the SSP expects.

In order to verify a partial SSP controller graph, $\mathcal{P}$, we translate it into a timed automaton (TA) model, $\theta(\mathcal{P})$. $\theta(\mathcal{P})$ is the product of a number of individual automata.

**Definition 3.2** [Timed Automaton [3]] A timed automaton $A$ is a tuple $\langle \mathcal{S}, s^i, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I} \rangle$ where $\mathcal{S}$ is a finite set of locations; $s^i$ is the initial location; $\mathcal{X}$ is a finite set of clocks; $\mathcal{L}$ is a finite set of labels; $\mathcal{E}$ is a finite set of edges; and $\mathcal{I}$ is the set of invariants. Each edge $e \in \mathcal{E}$ is a tuple $(s, L, \psi, \rho, s')$ where $s \in \mathcal{S}$ is the source, $s' \in \mathcal{S}$ is the target, $L \subseteq \mathcal{L}$ are the labels, $\psi \in \Psi_{\mathcal{X}}$ is the *guard*, and $\rho \subseteq \mathcal{X}$ is a clock reset. Timing constraints ($\Psi_{\mathcal{X}}$) appear in

guards and invariants and clock assignments. In our models, all clock constraints are of the form $c_i \leq k$ or $c_i > k$ for some clock $c_i$ and integer constant $k$. Guards dictate when the model *may* follow an edge, invariants indicate when the model *must* leave a state. In our models, all clock resets re-assign the corresponding clock to zero; they are used to start and reset processes. The state of a timed automaton is a pair: $\langle s, C \rangle$. $s \in \mathcal{S}$ is a location and $C : \mathcal{X} \rightarrow \mathbf{Q} \geq 0$ is a clock valuation, that assigns a non-negative rational number to each clock.

It often simplifies the representation of a complex system to treat it as a product of some number of simpler automata. The labels $\mathcal{L}$ are used to synchronize edges in different automata when creating their product.

A timed automaton *trace* is a series of state transitions that represents the computation of a timed automaton. Corresponding to any timed automaton, $A$, is a transition system, $S_A$, with two types of transitions: time-elapse transitions and jump transitions:

**Definition 3.3** [Time-Elapse Transition] A time-elapse transition, $\langle s, C \rangle \xrightarrow{t} \langle s, C+t \rangle$ can occur when for all $t'$ such that $0 \leq t' \leq t$, $t'$ satisfies the invariant $I(s)$.

**Definition 3.4** [Jump Transition] A jump transition, $\langle s_0, C \rangle \xrightarrow{e} \langle s_1, C' \rangle$, for some $e \in E$ can occur when $C$ satisfies the guard of $e$, $\psi(e)$ and $C'$ satisfies the reset of $e$ applied to $C$, $\rho(e, C)$.

**Definition 3.5** [Time Quotient] The time quotient of a timed automaton is a non-deterministic finite automaton whose states correspond to the locations of the timed automaton, and in which there is an edge $e$ between $s$ and $s'$ whenever there exists $s''$ and $t$ such that $s \xrightarrow{t} s'' \xrightarrow{e} s'$.

The CIRCA controller graph is the time quotient of a TA model of the corresponding controller. The translation of controller graphs to TA models is described in [5].

Figure 3 illustrates the timed automaton model corresponding to our running example, the partial Saturn orbit insertion controller shown in Figure 2. Since the SSP has not yet completed the controller in Figure 2, the timed automata model has sinks at locations 4 and 5, corresponding to the unplanned SSP states 6 and 8. Figure 4 illustrates the corresponding transition system reachability graph, where boxes correspond to a reachable location and clock region, represented as a difference bound matrix. The reachability graph shows that locations 4 and 5 are reachable, but since their corresponding states are unplanned, the verification traces halt there. The plan is safe so far, since failure is not reachable.

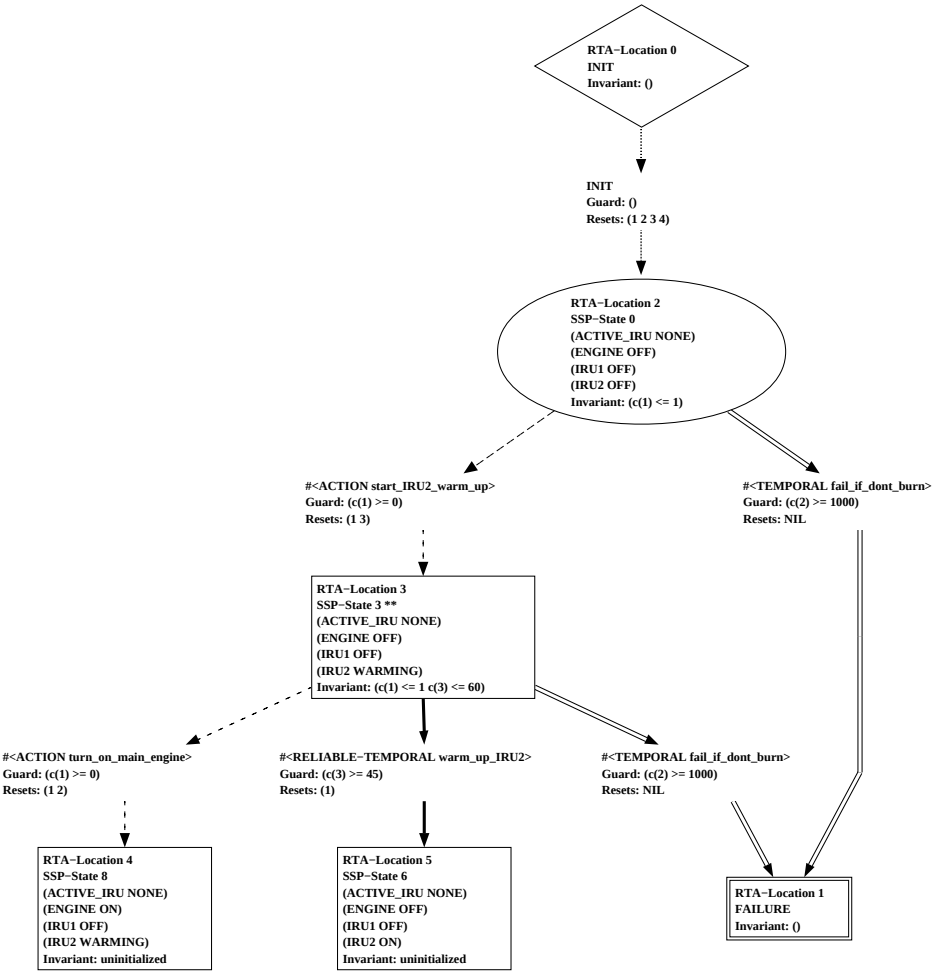


Fig. 3. The timed automaton model corresponding to Figure 2.

In this paper, we will concern ourselves primarily with *reachability verification* of a particular timed automaton. We will be asking if it is possible for a timed automaton to reach a particular location, $s \in S$. In particular, we will be checking the safety of a CIRCA controller by asking if a timed automaton corresponding to the controller can ever reach the distinguished failure state. While such reachability queries are not tractable, they are computable, and can be answered by simple graph search algorithms.

**Algorithm 2 (Reachability Verification).**

(i) *let openlist := initial state* $(\langle s^i, \mathbf{0} \rangle)$
(ii) *if openlist* $= \emptyset$ *then return* ***safe****;*
(iii) *let state := pop(openlist);*

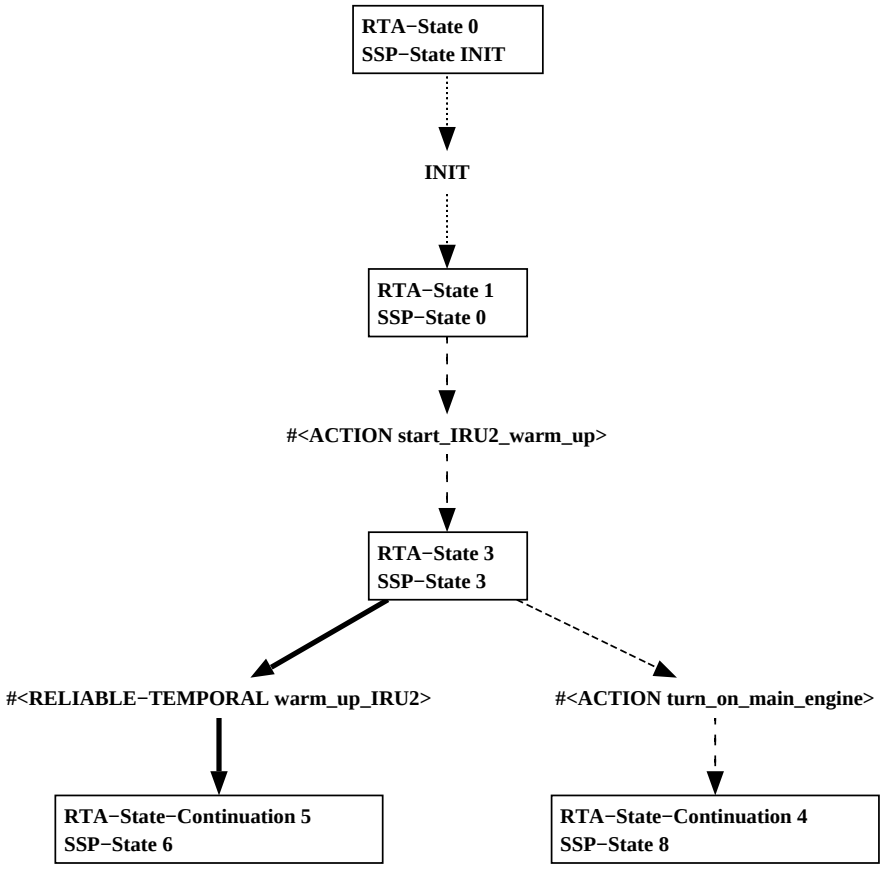


Fig. 4. The timed automaton reachability space corresponding to Figure 2.

(iv) *if visited(state) then goto ii;*
(v) *if failure(state) then return* ***unsafe****;*
(vi) *let succ := successors(state)*
(vii) *openlist := openlist* ∪ *succ;*
(viii) *goto ii;*

Of course, any naïve attempt to apply Algorithm 2 is doomed to failure. In particular, if one assumes dense time, the state space of this search may be uncountably large. Practical verification systems for timed automata typically search in a space of equivalence classes of states, since the state space of any timed automaton can be reduced to a finite number of equivalence classes[1]. Typically, a verification system will collapse together multiple states using clock *zones*. In the following discussion we will use "state" for both state and state equivalence class; no confusion should result since any practical

algorithm will have to manipulate the latter, rather than the former.

Verification systems also employ clever techniques for reducing the number of states that must be explored, answering the visited query (step iv, above), and computing the successor set (step vi). Furthermore, instead of simply returning **unsafe**, reachability verification systems typically return a *counterexample trace*, that exhibits a path from the initial state to the failure state, and can be used for debugging. To the best of our knowledge, CIRCA is unique in automating the exploitation of counterexample traces in controller synthesis (see [6] for an explanation of our technique for using counterexample traces to direct backtracking in controller synthesis search). We will return to the skeletal search algorithm later and describe modifications for our controller synthesis application.

Recall that the SSP verifies partial controllers during construction, before they are fully designed. Before the controller synthesis process is complete, there will be states that do not yet have action assignments. We verify partial controllers by treating such states as safe sink states. That is, we modify Algorithm 2 by adding a step after iv as follows:

4.5 if not action-assigned(state) then goto ii;

Note that when Algorithm 1 is completed, all of the states will have an action assigned to them, so the final verification will be a full verification. The sequence of verifications can be thought of as a fixpoint computation that converges on a full TA verification of the SSP controller.

## 4 Generating the State Space On-The-Fly

Even with the many techniques for efficient representation developed by timed automaton verification researchers, a direct combination of Algorithms 1 and 2 will suffer a state space explosion. We have developed two techniques to overcome this problem. The first, which we explain in this section is a special purpose way of lazily generating the timed automaton state space corresponding to a CIRCA controller. The second technique, which has never before been reported, and which we explain in the following section, is a way of reusing the results of the partial verifications executed in the course of controller synthesis.

One reason that CIRCA can efficiently search the space of the controllers generated in the course of Algorithm 1 is that it uses an implicit state-space representation. Instead of representing the state space explicitly, the state space is implicitly represented by the initial state feature set, and the set of transitions. CIRCA only assigns actions to states that it has reason to believe to be reachable. We have built a CIRCA-Specific Verifier (CSV) able to exploit

CIRCA's implicit state-space representation. The CSV constructs its timed automata, *both the individual automata and their product*, in the process of computing reachability.

The efficiency gains from our factored state representation come in the computation of successor states, step vi of Algorithm 1. A naïve implementation of the search would compute all of the locations (distinct discrete states) of the timed automaton up front, but many of those might be unreachable. We compute the product automaton lazily, on demand, thus constructing only reachable states.

The individual automata, as well as their product, are computed on-the-fly.

The transitions that synchronize with the primary transition are of three types:

(i) updates to the world automaton, recording the effect (the postconditions) of the primary jump on the discrete state of the world;

(ii) enabling and disabling jumps that set the state of uncontrolled transitions in the environment;

(iii) a jump that has the effect of activating the control action planned for the new state.

Accordingly, we can very efficiently implement a lazy successor generation for a set of states $S = \langle s, \mathbf{C} \rangle$, where $s$ is a discrete state and $\mathbf{C}$ is a symbolic representation of a class of clock valuations, in our case a difference-bound matrix. When one needs to compute the successor locations for the location $s$, one need only compute a single outgoing edge for the RTS transition and make one outgoing edge for each uncontrollable transition.

Making the outgoing edges is a matter of (again lazily) building the successor locations and determining the clock resets for the edge. The clocks that must be reset are: (a) For each uncontrolled transition that is enabled in the successor location, but not enabled in the source location, $s$, add a clock reset for the corresponding transition; (b) If the action planned for the successor location is different from the action planned for the source location, reset the action clock. These computations are quite simple to make and much easier than computing the general product construction.

We do not have room here to provide further details of the CSV. For a more thorough discussion, and for more details about the timed automaton semantics of CIRCA controllers, see [5]. In that paper we provide experimental results that illustrate the efficiency gains provided by the CSV.

## 5 Incremental Reachability Computations

As we explained earlier, in the course of controller synthesis, we repeatedly verify the partial controller automata that we generate. As one can readily imagine, this can be a very expensive process and, indeed, our experience was that the verification times dominated other aspects of controller synthesis. We have developed a technique that permits us to reuse the results of one (partial) reachability search in ensuing searches in the same controller generation.

The central insight behind the technique is that each run of the reachability search algorithm generates not only a judgment about the safety of the partial controller, but also a set of search states for each of the *frontier states* of the controller, where a frontier state is an as yet unplanned state that can be reached in one transition from a planned state. Indeed, if one looks at the partial verification state this way, one can see that a "safe" partial controller is a partial controller that can safely reach all of its frontier states.

In order to reuse search results, then, we need to save the states of the search at the frontier states as *continuations*. We make a table of such continuations, indexed by the states of the controller graph. Incorporating these continuations together with the modifications for partial verification described in the previous section, we get the following algorithm for verifying an action assignment to some controller graph state, $S$:

**Algorithm 3 (Incremental Reachability Verification).**

(i) ***var*** *continuations:* $f(S) \rightarrow$ *openlist* $:= \emptyset$*;*
(ii) ***var*** *cache_valid* $=$ ***false****;*
(iii) ***function*** *verify(S)*
(iv) *var openlist;*
(v) *if cache_valid then*
(vi) *openlist := continuations(S)*
(vii) *else openlist* $:= \{\langle s^i, \mathbf{0}\rangle\}$
(viii) *cache_valid :=* ***true****;*
(ix) *search(openlist);*
(x) *end verify*
(xi) ***function*** *search(openlist)*
(xii) *if openlist* $= \emptyset$ *then return* ***safe****;*
(xiii) *let state := pop(openlist);*
(xiv) *if visited(state) then goto xii;*
(xv) *if not action-assigned(state) then*
  (a) *continuations*$(SSP(state))$ *:= continuations*$(SSP(state)) \cup \{state\}$*;*
  (b) *goto xii;*
(xvi) *if failure(state) then return* ***unsafe****;*

(xvii) *let succ := successors(state)*
(xviii) *openlist := openlist ∪ succ;*
(xix) *goto xii;*
(xx) *end search*

Note that for every location, $l$, of a CIRCA timed automaton, there is a corresponding discrete state, `ssp(l)`, in the controller graph generated by CSM search. It follows from this that for every state explored in verification search, there is also a unique controller graph state that corresponds to it. With a certain abuse of notation, we refer to this as $SSP$(state) in Algorithm 3.

In Algorithm 3, we have described the continuations as functions that return a new openlist when "forced." For those of a more formal temperament, these continuations can also be thought of as functions from a state to an (eventual) value of **safe** or **unsafe**: $f(S) \rightarrow (f \rightarrow \{\textbf{safe},\textbf{unsafe}\})$. The continuations can be concretely implemented as either a hash table or an extensible array, depending on how the states are implemented.

In line a, we simply add the new state to the table of continuations for the corresponding frontier state of the controller graph $\mathcal{P}$, $SSP$(state). In practice, however, we check to make sure that we do not add any states that are subsumed by previously encountered ones.

When the search in the SSP backtracks, it invalidates the timed automaton implicit in Algorithm 3. Accordingly, when this occurs, when the SSP backtracks, we clear the cache. At the expense of some more book-keeping, we could reuse partial results even after backtracking. We do not believe that the search savings would be worth the additional overhead, but have not experimented extensively: the current savings are adequate to our current applications. However, we have a modified version of the controller synthesis algorithm that works in a gradually-refined abstraction space, and we have developed techniques to save results over model refinements. We do not have space to discuss those additional techniques here.

## 6 Evaluation

Not surprisingly, the benefits of incremental verification depend heavily on the degree to which the planner searches forward without backtracking, and the degree of temporal complexity in the domain (since the verifier is primarily addressing temporal correctness). If the planner's heuristic is generally correct and the planner does not need to backtrack often, the costs of incremental verification approach the optimal lower-bound, the cost of a one-shot verification after the plan is created. On the other hand, if the planner backtracks

frequently so that the verification trace cache is cleared frequently, the costs of incremental verification approach the worst-case upper bound represented by our earlier implementation, performing a complete from-scratch verification on each action decision.

Since the incremental verification does not impose significant additional storage or computation requirements beyond the original verification task, its worst case is not significantly worse than the complete-verification-every-time approach.

To evaluate the performance of our incremental verification approach, we have examined three problem sets. First, we examined the most benefit that incrementality can achieve, by testing with scalable robot tasking domains that are solved with little or no backtracking. Second, we assessed the overhead costs of the incremental approach on a spectrum of scalable domains that are constructed specifically to fool the search heuristic into poor choices, leading to extensive backtracking that clears the incremental verification cache. Third, we assessed the overall benefits of incremental verification on our regression testing domains of varying size, derived from real-world problems and prior investigations (available at
http://www.htc.honeywell.com/projects/sa-circa/).

### *6.1 No Backtracking*

One of the few available benchmark domains for this type of controller synthesis is based on a timed, discretized robot navigation and object-delivery problem [8,15]. The domain consists of a set of rooms connected by doors, a set of objects, and a robot that is capable of opening doors, moving from room to room through open doors, and picking up and putting down the objects. The goal is to move the objects from their initial positions into specified goal positions and to leave the robot in a predefined position. The domain differs from a classical closed-world form because it includes a random process (a "kid") that closes the doors. The size of the problem can be scaled by the number of rooms, "kid-doors," and objects to be delivered.

CIRCA solves problems in this domain without backtracking, because its greedy goal regression search heuristic always makes a safe choice that leads towards the goals, and there is no requirement that the resulting controller yield the shortest path to success. Since the system never backtracks, it is always able to simply extend its verification traces incrementally.

To evaluate incremental verification in this domain, we generated 300 test problems in an 8-room, 7-door world. We varied the number of objects to be delivered from one to six, with randomly-generated initial and goal rooms. We varied the number of randomly-selected "kid-doors" from zero to four. For

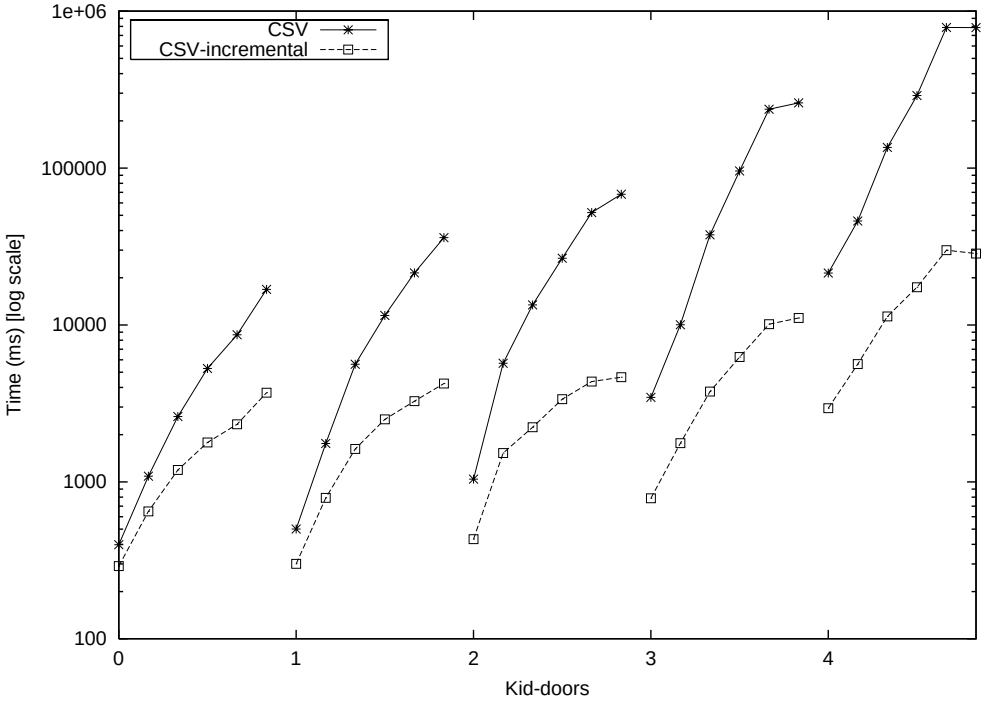


Fig. 5. When the CSM does not backtrack, incremental verification saves up to 97% of planning time.

each setting of the number of delivery goals and kid-doors, we ran the CSM on ten sample domains [4].

Figure 5 plots the average controller synthesis time for each set of ten domains. In every one of the 300 tests, the incremental version is faster than the non-incremental version, saving up to 97% of the overall controller synthesis time. The savings increase as the problems become more complex, since the non-incremental version runs an exponential verification process after each decision in the growing search, while the incremental version performs only a linear expansion of its cached verification traces.

### *6.2 All Backtracking*

To evaluate the overhead costs of incremental verification in domains where extensive backtracking largely negates its advantages, we created a different set of scalable domains and degraded the CIRCA heuristic, so that the search engine makes a poor decision at every state the first time it is encountered. The domains are composed of a series of $N$ actions that are required to achieve the

[4] Note the robot is also given a random final destination to park at, so the number of goals varied from two to seven.

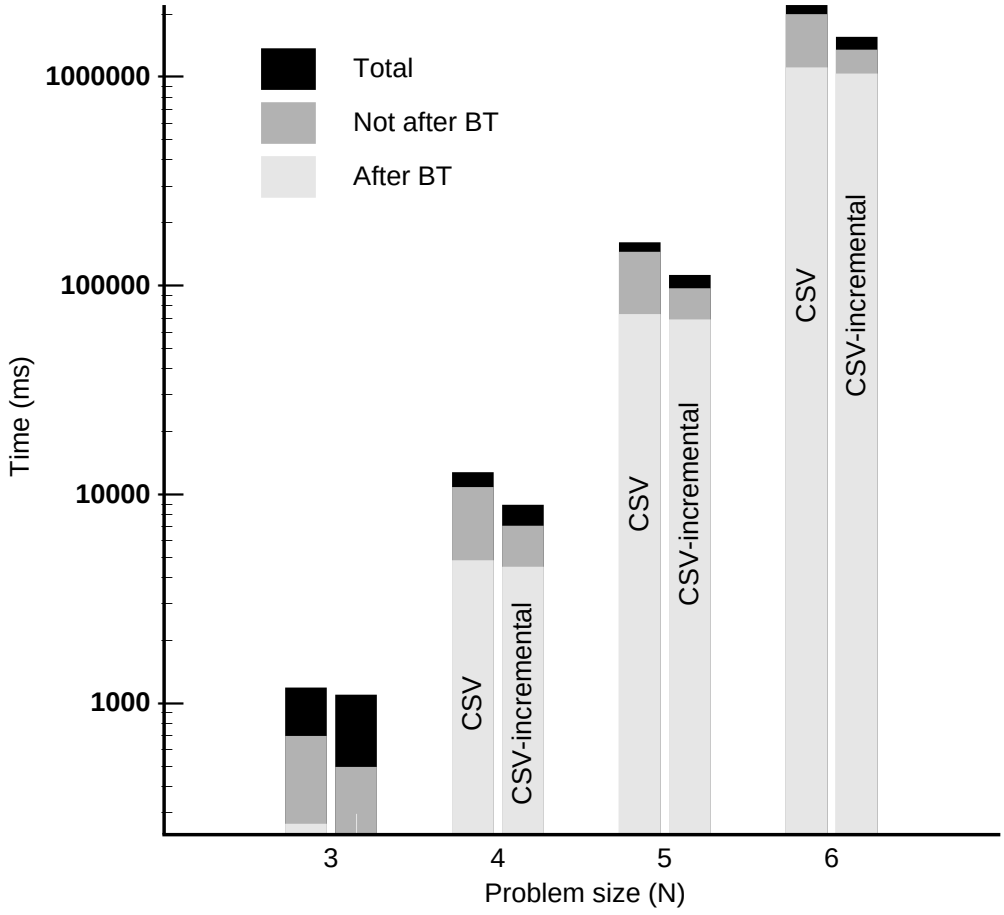


Fig. 6. Even when backtracking is prevalent, incremental verification requires minimal overhead and provides significant benefits.

final goal, and they must be executed in a specific order or else uncontrollable temporals will occur that undo the progress made towards the goal. We declare the final goal to be mandatory, so that CIRCA is not allowed to build a plan that does not achieve that goal. We carefully construct the domain so that the degraded heuristic always suggests the wrong ordering first, forcing backtracking in proportion to $N!$.

Figure 6 illustrates the resulting performance for values of $N$ from three to six. As the number of actions in the ordered chain grows, the degraded heuristic forces the controller synthesis system to explore many more incorrect orderings (infeasible controllers that make failure reachable). For example, the

$N = 6$ domain causes 76657 backtracks, each of which clears the verification cache. The bottom segments in Figure 6 show that the overhead of the incremental verification algorithm is minimal, so that the system uses about the same amount of time performing verifications "from scratch" (after backtracks) in either mode. And even in this backtrack-laden domain, incremental verification yields significant benefits.

Note that these performance results are obtained with a degraded heuristic; with the normal CIRCA heuristic, these domains are all solved without backtracking, in less than one second. These degraded-heuristic experiments only serve to show the worst-case penalty of incremental verification.

### *6.3 Regression Suite*

In a set of 23 domains from our regression suite, our evaluations indicate major advantages for incremental verification in moderate and large domains. These domains range from very small test-cases that exercise planning behaviors such as backtracking, to fairly large domains drawn from models of robotic workcells, spacecraft commanding, and UAV tasking.

As shown in Figure 7, the incremental CSV algorithm uses less time than the batch CSV in all cases except two problems: one short problem where their times are essentially equal and one that ran until the test's 20-minute timeout. For reference, the chart also shows the CSM runtimes when using the Kronos verifier tool, which also operates in a batch mode (and over a file-system based interface). The Kronos results are not always perfectly comparable, since in some cases Kronos returns different culprit paths than CSV, which leads to different backjumping behavior and different search space exploration. However, in all cases the CSV and CSV-incremental algorithms are faster. It is worth noting that Kronos is a far more general verification tool than CSV, which is tailored to our specific reachability concerns.

## 7 Related Work

The SSP algorithm is closely related to techniques for game-theoretic synthesis of controllers for timed systems [2,10]. One difference is that the SSP algorithm works "on-the-fly" starting from an initial state and building forward by search. The game-theoretic algorithms, on the other hand, use a fixpoint operation, to be implemented by dynamic programming, to find a controllable subspace, starting from unsafe states (or other synthesis failures).

Tripakis and Altisen [19] have independently developed an algorithm very similar to ours. They use the term "on-the-fly" for algorithms that generate their reachable state spaces at the same time as they synthesize the controller.

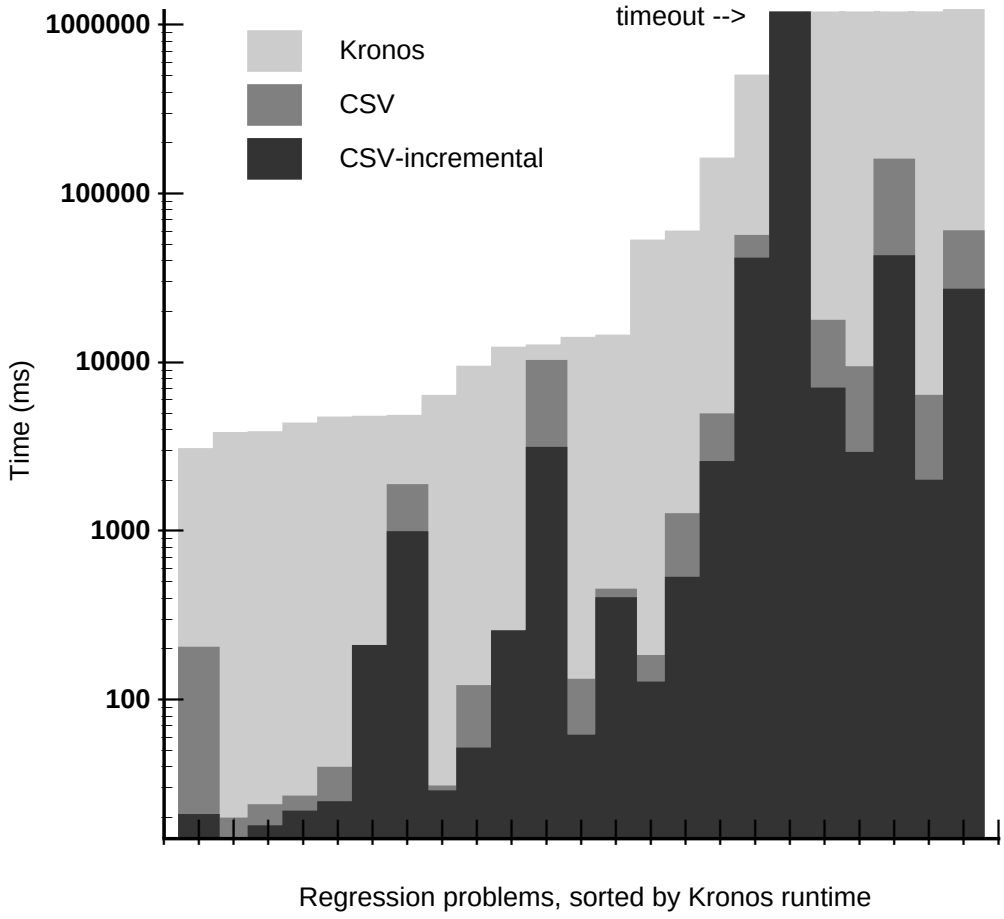


Fig. 7. Our spectrum of regression tests also show significant benefits from incremental verification, averaging 40% savings over batch CSV.

Most AI planning algorithms, including CIRCA's [11,12] are on-the-fly in this sense. We believe some of our efficiency improvements could be adapted to their algorithm.

Sahay *et. al.* [17] describe algorithms for modifying a "timed state-variable graph" (TSG) analogous to the region graph built during CIRCA plan verification. They modify the TSG when changes are made to the corresponding system design. Their solution is aimed at a different problem, where full verifications of full designs are run, rather than to verification in on-the-fly controller synthesis.

Gordon-Spears [7] has also recognized the need for swift verification of

plans for autonomous systems. She proposes novel "incremental reverification" algorithms suitable for use when a machine learning system adapts an existing plan. Unlike CIRCA's controllers, hers are *untimed* automata, which reduces the computational complexity of verification. However, because Gordon-Spears' agents adapt their plans in ways that may affect large portions of the plan, in the worst case the entire plan will need to be re-verified; this makes her verification problem more like Sahey *et al.*'s.

## 8 Conclusions and Future Directions

We have described an incremental verification technique that can provide major benefits for on-the-fly controller synthesis and verification. While our approach is implemented in the context of the CIRCA system, it could be more broadly applied to verification processes in other automatic and manual system design applications. Our evaluations have demonstrated that the approach induces little or no additional overhead, but can eliminate up to 97% of the verification time consumed by a normal batch verification process used in iterative fashion.

In our current implementation, the cache of verifier traces is cleared any time the SSP backtracks to change one of its decisions. In theory it would be possible to always perform incremental verification, by keeping track of the detailed dependencies between SSP decisions and the verifier traces, and only invalidating those traces affected by altered SSP decisions. We have not yet tried to build this more complex version of the approach, because preliminary estimates indicate that it is not likely to yield a huge payoff. The remaining complexity lies primarily in the search heuristic and in the need to verify a successful plan at least once, which is unavoidable in our framework.

## References

[1] Alur, R., *Timed automata*, in: *NATO-ASI Summer School on Verification of Digital and Hybrid Systems*, 1998.

[2] Asarin, E., O. Maler and A. Pneuli, *Symbolic controller synthesis for discrete and timed systems*, in: *Proceedings of Hybrid Systems II*, Springer Verlag, 1995 .

[3] Daws, C., A. Olivero, S. Tripakis and S. Yovine, *The tool Kronos*, in: *Hybrid Systems III*, 1996.

[4] Gat, E., *News from the trenches: An overview of unmanned spacecraft for AI*, in: *AAAI Technical Report SSS-96-04: Planning with Incomplete Information for Robot Problems*, 1996.

[5] Goldman, R. P., D. J. Musliner and M. J. Pelican, *Exploiting implicit representations in timed automaton verification for controller synthesis*, in: *Proceedings of the 2002 Hybrid Systems: Computation and Control Workshop*, 2002.
URL http://www.cs.umd.edu/users/musliner/papers/hybrid02.ps.Z

[6] Goldman, R. P., M. J. S. Pelican and D. J. Musliner, *Guiding planner backjumping using verifier traces*, in: *Proc. 14th Int'l Conf. on Automated Planning and Scheduling*, 2004.

[7] Gordon, D. F., *Asimovian adaptive agents*, JAIR **13** (2000), pp. 95–153.

[8] Kabanza, F., M. Barbeau and R. St.-Denis, *Planning control rules for reactive agents*, Artificial Intelligence **95** (1997), pp. 67–113.

[9] Khatib, L., N. Muscettola and K. Havelund, *Mapping temporal planning constraints into timed automata*, in: *Eighth Int'l Symp. on Temporal Representation and Reasoning*, 2001.

[10] Maler, O., A. Pneuli and J. Sifakis, *On the synthesis of discrete controllers for timed systems*, in: *STACS 95: Theoretical Aspects of Computer Science*, Springer Verlag, 1995 pp. 229–242.

[11] Musliner, D. J., E. H. Durfee and K. G. Shin, *CIRCA: a cooperative intelligent real-time control architecture*, IEEE Trans. Systems, Man, and Cybernetics **23** (1993), pp. 1561–1574.

[12] Musliner, D. J., E. H. Durfee and K. G. Shin, *World modeling for the dynamic construction of real-time control plans*, Artificial Intelligence **74** (1995), pp. 83–127.
URL http://www.cs.umd.edu/users/musliner/papers/musliner-aij.ps.Z

[13] Musliner, D. J. and R. P. Goldman, *CIRCA and the Cassini Saturn orbit insertion: Solving a prepositioning problem*, in: *Proc. NASA Workshop on Planning and Scheduling for Space*, 1997.

[14] Musliner, D. J., J. A. Hendler, A. K. Agrawala, E. H. Durfee, J. K. Strosnider and C. J. Paul, *The challenges of real-time AI*, IEEE Computer **28** (1995), pp. 58–66.

[15] Pistore, M. and P. Traverso, *Planning as model checking for extended goals in non-deterministic domains*, in: *Proc. Int'l Joint Conf. on Artificial Intelligence*, 2001, pp. 479–486.
URL http://citeseer.nj.nec.com/pistore01planning.html

[16] Ramadge, P. J. and W. M. Wonham, *The control of discrete event systems*, Proc. of the IEEE **77** (1989), pp. 81–98.

[17] Sahey, A., J. J. Tsai and A. P. Sistla, *An incremental verification algorithm for real-time systems*, Int'l Journal of Software Engineering and Knowledge Engineering **9** (1999), pp. 203–216.

[18] Simmons, R., C. Pecheur and G. Srinivasan, *Towards formal verification of autonomous systems*, in: *Proc. Conf. on Intelligent Robots and Systems*, 2000.

[19] Tripakis, S. and K. Altisen, *On-the-fly controller synthesis for discrete and dense-time systems*, in: J. Wing, J. Woodcock and J. Davies, editors, *Formal Methods 1999*, number 1708 in Lecture Notes in Computer Science, Springer Verlag, 1999 pp. 233–252.